

CONFIDENTIAL

TO: AMIGA neophytes

FROM: Greg Riker
Project Leader
Artist Work Station

RE: AWS68K Overview

DATE: 9 January 1985

Greetings. If you're reading this, I presume that you have been annointed to carry forth the Electronic Arts logo to the Amiga and beyond. Congratulations on being a pioneer in an exciting new era for home computing. I have been working with the Amiga 'hands-on' for about 2 months now, and I assure you that this box will not become obsolete in a year. Instead, it is packed with functionality that is rare in a micro. There is 128k of densely coded ROM in the box, which means lots of System support for common tasks. At times, I find the sheer bulk of the information necessary to make the box do something overwhelming. However, all of the information is vital to the operation of the machine - there's just a lot of it. Those of you who waded through documentation from Atari in the early days will get the same type of flavor and content from what's available.

You probably had quite a bit of 6502 coding experience before joining forces with EA. That means that you probably had your own set of tools for 6502 development - a fast assembler, a flexible editor, and reasonable debugging tools. We at Electronic Arts have found that most of you use different combinations of tools. Just think of how much experience you could barter if everyone had used the same set of tools from the beginning! Well, friends, for better or for worse, we're trying to establish a 'standard' set of tools for use on the Amiga. Loosely packaged under the name 'Artist Work Station 68k', we have picked the 'must-have' tools from our infinite wish list, and begged, borrowed, bought or wrote the components necessary to have a minimal, yet functional, set of tools. Please keep in mind that you are among the first to use this combination of tools in an environment that until recently, was largely theoretical. We NEED your assessment of the system to fold improvements back into the system. Please don't be afraid to be critical when criticism is needed. Many of the observations about its predecessor, the AWS6502, went into the ingredients for this version, and thus AWS68k has lots of neat features that AWS6502 didn't have. Your input counts.

CONFIDENTIAL

Specifically, the Software components include

1) The Editor.

VEDIT is a full screen, programmable, user-configurable text editor. It's not much for formatted word processing, but it's nice for programming. You may find VEDIT frustrating at first, but I encourage you to give VEDIT a couple days before you give up on it. If you still feel like you've got to use something different, do it.

2) The Cross-Compiler

Amiga has contracted Lattice to make 3 versions of their 68k Compiler -- UNIX, IBM PC, and native. Lattice is the official sanctioned C for the Amiga. The version that you will be using is a Beta version.

3) The Assembler/Linker

We're currently using the Quelo assembler/linker, basically because it came with the beta copy of the C compiler. This is a fine assembler/linker package, but it's not the one that Amiga will be supporting. A company called MetaComCo has been contracted to provide an assembler/linker that handles overlays. It will be available in the same three environments as the C compiler. We will let you know when it's time to switch over.

4) The Downloader

This is a new NWIRE downloader that runs much faster than the one included with AWS6502. As a matter of fact, for those of you using AWS6502, you can use the same downloader included with this package to get faster downloads to 6502 targets.

5) The Debugger

The mind control device for your Amiga, a leash for your micro, yes fans, DDT returns in its THIRD incarnation as a debugger. Thanks and a tip of the hat to Jim Dunion for his unselfish help in making the mutant versions of his original native Atari debugger come to life.

6) Prism

Dan Silva's wonderful bit map editor includes support for the Amiga now, so you can edit pictures on your PC and download them to the Amiga. If you have a Tecmar board for the PC, you can get 320x200 resolution with 16 colors on the PC, and then see the same picture on the Amiga.

CONFIDENTIAL

These six components make up the guts of AWS68k. Pretty soon, we plan to have some other tools for you to add to the list, including a sound editing utility. In the beginning, you're on your own. If you find anything that makes a valuable contribution to the synergy between these tools, let me know about it. For instance, I've been thinking that SideKick might be a neat addition to the package -- you could edit your source while you're in the debugger! And of course, there's always that Hex calculator.

Reading Recommendations, etc.:

I absolutely recommend three additions to your library.

- * The C Programming Language - Kernighan & Ritchie
(Prentice-Hall)

Everybody says you ought to have this one, since the guy who designed C is a co-author. Kind of bland, but a goldmine of subtlety concerning syntax that you'll go back to again and again.

- * C Primer Plus - Waite, Prata and Martin
(Sams)

This book is packed with examples of well-designed C code. I find it most helpful when I am trying to learn a new feature of the language. At times, they get carried away with their delight with their own humor, but the book is well done.

- * 68000 Assembly Language Programming - Kane, Hawkins, Leventhal
(Osborne - McGraw Hill)

This book in prominent display on your bookshelf makes it clear to all that you are a serious 68000 programmer. Mostly useful for forgetting 6502 or 8086 mnemonics.

I absolutely recommend one addition to your Software collection

- * Instant C - Rational Systems (\$425-\$500)
Available through your producer or Rational Systems

This C interpreter package will let you learn C real fast on your AWS. By eliminating the compiler-linker steps from your development loop, you can get to the algorithm that you're looking for very quickly. I have found this package to be indispensable for learning some of the subtler aspects of C.

CONFIDENTIAL

Assembler vs C:

Most of you have made your mark in the software world writing pure assembler code. I offer you an alternative view of the future. If you consider C as assembly language macros, you'll start to appreciate its position as the Official Language of the Amiga. C provides you with a barebones set of functions like for-next loops, do-while loops, logical and boolean operators, and other necessary low-level stuff. After you understand the power of C, if you have a particular function that you want to write in assembler, go right ahead. The interface from C to assembler is very clean and easy to implement. Amiga chose to define all of the ROM calls in terms of C style calling descriptions. Once you get used to working in C, you'll thank them for their decision. If you feel yourself slipping towards assembly as your first choice, give your software buddy at EA a call and see if we can help you. Your productivity will absolutely increase as you become fluent in C. The demos that are being provided to you along with your startup kit were written entirely in C - no assembler.

The 'Standard' software tools:

The combination of tools that has been described comprises the nucleus of what Amiga will support as 'official' software tools. Therefore, it is pretty important that you stick with these tools for the near future. There undoubtedly will be new/better/improved alternatives to the individual components, but you will use them at your own risk. We'd really like to foster an atmosphere of cooperative learning on this machine, and the best way to do that is for all of us to share the same environment. Let me know if you've found a better solution, and we'll try to incorporate it if practical.

HELP!!! :

If you have problems, please feel free to contact me. I will attempt to solve your problem as rapidly as possible, or direct you to the person who is most likely to be able to solve your problem. I can be reached from 8:30 am - 2:30 pm local time at 415-571-7171.

CONFIDENTIAL

The 'Complete' package:

A complete AWS 68k environment consists of the following list. Make sure that you have all of the components before you start to do anything serious.

- 1) Your host machine, an IBM PC or approved equivalent
- 2) Your Amiga with a color monitor (analog RGB preferred)
- 3) An EA-modified AST 6 pak board
- 4) NWIRE cables for IBM --> Amiga
- 5) Five IBM disks
 - #1 - AWS 68k Boot Disk
 - #2 - AWS 68k Runtime Disk
 - #3 - PRISM
 - #4 - AWS 68k Sample Data Disk
 - #5 - Library object code modules
- 6) One Amiga disk with NWIRE Target Code
- 7) A packet of memos, covering:
 - #1 - Bootstrapping the AWS 68k Environment
 - #2 - Building the Sample C File, SIMPLE.C
 - #3 - Using DOWNLOAD and DDT
 - #4 - Using PRISM
 - #5 - Printouts of the DDT Help Screens
- 8) An RS-232 cable for connecting AWS --> Amiga for bootstrap

A firm handshake . . .

To start up your system, read the memo entitled 'Bootstrapping the AWS 68k Environment'. Once you have started the Amiga, proceed to 'Building the sample C file, SIMPLE.C'. Then, move on to 'Using DOWNLOAD and DDT'. If you've gotten this far, you're well on your way to becoming a Amiga software artist.

Once again, congratulations on being selected to create an Amiga product for Electronic Arts. We look forward to a monster hit from you. Go forth and code.

TO: Amiga Artists
FROM: Greg Riker
DATE: 9 January 1985
RE: Bootstrapping the AWS 68k Environment

CONFIDENTIAL

In order to bootstrap your Amiga, follow this sequence :

- 1) Place the 'AWS 68k Boot Disk' in Drive A:. If the power is off, turn on your AWS. If the power is already on, press Ctrl-Alt-Del to reboot your AWS.
- 2) After the boot sequence has finished, type :
 - 'a' to make a: the default drive.
 - 'cd pc-talk' to enter the directory containing PC-TALK.
 - 'pc-talk' to run PC-TALK.
- 3) A full screen of PC-TALK text appears, followed by a prompt at the bottom that says 'press any key . . .'. Press the space bar to continue.
- 4) The screen clears, and some information about PC-TALK appears in the upper right hand corner. Press ALT-C to clear the screen. The default parameters that PC-TALK uses are correct for the Amiga as of this writing. PC-TALK is configurable to other settings if necessary. Call me if you need help with this.
- 5) Verify that your Amiga and your AWS are connected with an RS-232 cable through the serial ports. On the Amiga, there is a DB-25 port labeled 'UART'. On the AWS, the DB-25 port on the AST 6-Pak board is the serial port.
- 6) Verify that your NWIRE cable is connected between the AWS and Amiga.
- 7) Power up the Amiga. You will see a message on the PC-TALK screen from the Amiga, there will be a pause, the screen will turn to blue and black stripes, and then more text from the Amiga will appear on the PC-TALK screen. The environment that you are talking to is the ROM debugger in the Amiga.
- 8) Type 'g c ff0000' (no spaces, added here for clarity). This tells the Amiga to pass control to the internal MiniFileSystem. Insert your Amiga Boot Disk into the disk drive, and close the door.
 - Type 'b' to boot the disk.
 - Type 'ls' to see the files on your Amiga Boot Disk.
 - Type 'nwire' to load the NWIRE Target Code into memory.
 - Type 'q' to exit the MiniFileSystem.

CONFIDENTIAL

IMPORTANT NOTE: This is NOT the file system that will be delivered for the system. It is a temporary solution that allows developers to keep code and data on a local disk. It is designed for bootstrap and demo purposes only. The real file system will be just that - a real file system. Don't worry about learning all of the ins and outs of this one.

- 9) You are now back in the ROM Debugger.

Type 'gc7f000' to start the AWS 68k target code.

The AWS 68k Target code is now executing. You won't see any confirmation from the Amiga, because the AWS 68k Target Code is in control now. If you do see some sort of a message, you've probably blown it somewhere along the line. Start over at step 7.

- 10) If all is well :

Type 'ALT-X' to exit PC-TALK

Type 'y' to confirm that you really want to leave
PC-TALK.

- 11) MS-DOS reloads COMMAND.COM from drive A:, and you're ready to talk to your Amiga via DOWNLOAD and DDT.

CONFIDENTIAL

TO: Amiga Artists
FROM: Greg Riker
RE: Building the sample C file, SIMPLE.C
DATE: 9 January 1985

This is a summary of how to build an executable image of SIMPLE.C. You will find all of the files necessary to build SIMPLE on the disk entitled 'AWS 68k Sample Data Disk'.

- 1) Make sure that the 'AWS 68k Runtime Disk' is in Drive A:.
- 2) Make sure that the 'AWS 68k Sample Data Disk' is in Drive B:.
- 3) Make the default drive C: by typing 'c/'
- 4) Copy SIMPLE.C and SIMPLE.LNK from drive B: to drive C:.

| | |
|--------------------------|--|
| Type 'copy b:simple.c' | to load the C source code for SIMPLE |
| Type 'copy b:simple.lnk' | to load the linker control file for SIMPLE |
- 5) Type 'xc simple' to invoke the cross-compiler

This step will take about two minutes. You can monitor the progress of the cross compilation by watching the various messages that come up on the screen. Obviously, SIMPLE is more than a few lines of code! At the end of the cross-compile, SIMPLE.LTX is backed up to drive B:. Files ending in .LTX are object code modules output by the cross-compiler or assembler.
- 6) Type 'copy b:*ltx' to load the object modules for SIMPLE

These files correspond to the contents of the .LNK file. They comprise the Amiga ROM interface code, and some C library functions.
- 7) Type 'link simple' to begin the link process

If all goes well, you'll have no errors from the link, and you will be told that SIMPLE.SYM and SIMPLE.S28 have been backed up to drive B:.

CONFIDENTIAL

8) Type 'dir simple' to see all of the SIMPLE files

You'll see five files :

| | |
|------------|-------------------------------|
| SIMPLE.C | the original C source code |
| SIMPLE.LNK | the Linker Control File |
| SIMPLE.LTX | SIMPLE's object code module |
| SIMPLE.SYM | the symbol table for SIMPLE |
| SIMPLE.S28 | the executable code of SIMPLE |

9) You have just built a program!

10) Go to the memo entitled 'Using DOWNLOAD and DDT' to run the demo.

CONFIDENTIAL

TO: Amiga Users
FROM: Greg Riker
RE: Using DOWNLOAD and DDT
DATE: 9 January 1985

Before you can download or debug a file, the Amiga must have been bootstrapped so that the AWS 68k Target Code is currently running on the Amiga. See the memo titled "Bootstrapping the AWS 68k Environment" for instructions on how to bootstrap the Amiga.

You must also have a .S28 file to download. For this demo, I assume that you have built the SIMPLE.S28 file by following the instructions in the memo titled "Building the Sample C File, SIMPLE.C".

- 1) Verify that the 'AWS 68k Runtime Disk' is in Drive A:.
- 2) Verify that the files SIMPLE.S28 and SIMPLE.SYM are on Drive C:.
- 3) Verify that Drive C: is the default drive.
- 4) Type "download simple.s28" to download the executable code

You will see the addresses to which the downloader is loading, then a line of dots. Each dot represents one line of the S28 file. Note the Run Address that is specified at the end of the download -- this is the Run Address (where 'main' is) in your C code.

- 5) Type "ddt simple.sym" to invoke DDT with the symbol table from SIMPLE.C

The DDT screen will come up, and the Main Window will show code disassembled in ROM. This is the re-entry point for the ROM debugger.

- 6) Type "?" to see a summary of the commands and their syntax
- 7) Type "B" to learn more about breakpoints
- 8) Type <space bar> to return to the disassembly window.
- 9) Type "E 'main'" to look at the code at 'main'
- 8) Type "R PC,2005A" to set the PC to the address of 'main'.

CONFIDENTIAL

- 9) Use the cursor keys to page up and down through the code. You will see lots of calls to interesting sounding routines. Get familiar with the effect of each cursor control key, including PgUp and PgDn.
- 10) Type "w" to toggle the display window to HEX & ASCII
- 11) Type "w" to toggle back to the disassembly window
- 12) Type "E *" to return the window to the current PC
- 13) Type "+", using the large "+" key at the extreme right side of the keyboard. One instruction will execute, and the screen will update with the new values. Single step 3 more times until you get to the JSR OpenLibr instruction.
- 14) Type "P" to treat the next instruction as a Procedure.

The PROCedure instruction sets a breakpoint after the current instruction, and then runs the code at the current PC. This is just what you want to treat JSR's as a single instruction. You may use "P" anywhere you would use "+", but note that using a "P" command at a Bxx instruction will not give you control back until execution reaches the next instruction past the Bxx instruction. This is a good way to get out of a loop quickly.

- 15) Look at the listing for SIMPLE.C. On the second page, just under 'main0' is a C instruction:

```
GfxBase = (struct GfxBase *) OpenLibrary("graphics",0);
```

Look at the DDT screen. The instruction at 20064 reads :

```
PEAL $20F4C
```

Type 'E 20F4C', and then change the display window to the HEX/ASCII mode. In the ASCII window, there is a string, "graphics". This is an argument to the OpenLibrary function call from C. When you see a JSR to a named routine in the disassembly window, you can usually correlate it to a line of C source code. The next JSR instruction at address 20084 is to a routine called LoadCirs. If you look at the C source code, you'll find that C instruction 2 lines after the call to OpenLibrary.

- 16) Type "G *" to run the code at the current PC.

You should see a yellow "O" from the EOA logo bouncing around on the screen. On the DDT screen, in the command window, you will see a message that tells you to press any key to stop execution in the target.

- 17) Type <space bar> to stop the Amiga

The "O" figure may or may not be displayed when you stop the machine, depending on where the code was when it was interrupted.

- 18) Type "T" to invoke the Trace Options menu

Use the arrows and space bar to highlight the following options:

Instruction Window
Stack
Registers

Type <CR> to accept the highlighted options and return to the disassembly screen.

- 19) Type 'T' to run the code Interpretively, displaying the trace options that you selected in step 12.

- 20) You will see the code executing an instruction at a time, with screen updating the Instruction Window, the Stack, and the Register display after each instruction. At this speed, you won't see Amiga screen activity very often.

- 21) Press any key to stop Interpreted execution.

- 22) Type "T" to invoke the Trace Option menu

This time, select High Speed. High Speed mode is mutually exclusive with all of the other display mode options, so they will be turned off automatically for you. Press <CR> to accept and return to the disassembly window.

- 23) Type "T" to run the code in the High Speed Interpreted mode.

If the ball is moving at normal speed, the High Speed invocation didn't 'take'. Press any key to interrupt the target, then press "T" to restart. Keep doing this until the ball is moving very slowly, appearing and disappearing on the screen. For an explanation of this phenomenon, read on. If you're not interested, skip forward to step 24.

How the High Speed Mode works:

The 68000 has a mode called the Trace Mode, which is invoked by setting the Trace Bit (bit 15) in the Status Register. On the

DDT screen, this bit is shown with the letter 'T' on the far left of the status bit display. When the Trace bit is set, execution is interrupted after each instruction, and control is passed through a vector to a routine that causes a variable delay. Pressing the left and right arrow keys causes DDT to actually write a new delay value into the routine. In this fashion, you can control the effective execution speed of the machine over a wide range. Occasionally, when you invoke the High Speed mode, it doesn't 'take'. This occurs when you interrupt the machine in the middle of an interrupt routine. DDT faithfully sets the Trace bit, but when the end of the interrupt routine comes along, the status registers are popped off the stack, clearing the Trace bit that was just set! I don't have a foolproof solution for this problem, so follow the procedure outlined above -- stop the target, and reissue the "T" command until the speed slows. I find that this rarely takes more than 3 attempts.

24) Type "<-" and "->" to change the execution speed

25) Type <space bar> to stop the target

26) Now, we will use the mini-symbol table feature of DDT.

Type "E 'vx'" to examine the X velocity component of the movement.
Type "E 'vy'" to examine the Y velocity component of the movement.

Note the addresses of these variables. You can track the contents of these variables with the M command, which Monitors locations in Target memory.

Type "M 2,xxxxxx<CR>" to tell DDT the address of the word variable 'vx'.

The cursor will go the top line of the mini-symbol table, where DDT is waiting for you to type in the name of the variable that we're monitoring.

Type "VX" to name it "VX"

In a similar way, monitor the location of 'vy'.

Type "M 2,xxxxxx<CR>"
Type "VY"

27) In the far right column labeled 'Value', the current contents of VX and VY are displayed in word format. To change the velocity components :

| | |
|-------------------|--------------------------|
| Type "E 'vx'" | to position the window |
| Type "D 0020<CR>" | to change the x velocity |
| Type "G *" | to watch the new speed |
| Type <space bar> | to stop the target |
| Type "E 'vy'" | to position the window |
| Type "D 0001<CR>" | to change the y velocity |
| Type "G *" | to watch the new speed |

28) Well, you've done it. You have built an executable C image on your AWS, downloaded it into the Amiga, and controlled the execution of it. Now, it's up to you to be brilliant.

CONFIDENTIAL

```

/* simple.c: One moving bob colliding with walls. */
#define BORDERHIT BNDRY_HIT
#include <std.h>
#include <audio.h>
#include <blit.h>
#include <collide.h>
#include <copper.h>
#include <display.h>
#include <dma.h>
#include <gels.h>
#include <gfx.h>
#include <gfxblit.h>
#include <gfxmacros.h>
#include <exstruct.h>
#include <graph.h>
#include <gfxbase.h>
#include <int.h>
#include <regs.h>

#define SHEIGHT 28      /* vSprite height */
#define SWIDTH 2        /* vSprite width */
#define SDEPTH 1        /* vSprite depth */

#define WIDTH 320        /* screen width */
#define HEIGHT 200       /* screen height */
#define DEPTH 4          /* screen depth */

#define FOREVER for(;;)

/* general usage pointers */
struct GfxBase *GfxBase;      struct RastPort *writePort = 0;
struct View v;                struct RastPort rP;
struct ViewPort vp;           struct IVPArgs viewArgs;
struct GelsInfo gelsinfo;     struct vSprite dumSprite[2] = {0};

struct vSprite sprite = {0};  /* vSprite info for the sphere. */
struct bob bob = {0};         /* Bob info for sphere.*/
int vx, vy;                   /* Sphere's velocity.*/
short image0[SDEPTH*SHEIGHT*SWIDTH] = { /* sphere image.*/
0,      0,      0,      0,      0x003F, 0xFF00, 0x01FC, 0x0,
0x0,     0x0,     0x07FF, 0xFFFF, 0x1FC0, 0x0,     0x0,     0x0,
0x3FFF, 0xFFFC, 0x7FC0, 0x0,     0x0,     0x0,     0x7FFF, 0xFFFE,
0xFFC0, 0x0000, 0x0,     0x0,     0xFFFF, 0xFFFF, 0xFFE0, 0x0000,
0x0,     0x0,     0xFFFF, 0xFFFF, 0x7FF8, 0x0000, 0x0,     0x0,
0x7FFF, 0xFFFE, 0x3FFF, 0x0000, 0x0,     0x0,     0x1FFF, 0xFFFF,
0x07FF, 0xF000, 0x0,     0x0,     0x01FF, 0xFF00, 0x003F, 0xFC00 };

short colors[32] = { /* Each color is: 0x-blue-green-red */
0x0000, 0x0ff, 0x000, 0x000, 0x000, 0x000, 0x000, 0x000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0xffff,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0xffff };

/* Copy array contents: long destination[nLongs] = source[].*

```

```
CopyOfLongs(destination, source, nLongs)
    register long *destination, *source;
    register int nLongs; {
do { *destination++ = *source++; } while (--nLongs); }
```

CONFIDENTIAL

```
/* routine called when a border collision is detected */
borderPatrol(s, b)    struct vSprite *s;    int b; {
if (b & (TOPHIT | BOTTOMHIT))    vy = -vy;    /* Reverse direction.*/
if (b & (LEFTHIT | RIGHTHIT))    vx = -vx; }
```

```
main() {
register int i;
struct GelsInfo *gi;
long *sptr;
char *pptr[DEPTH], *rptr[DEPTH];
short nextlines[8], *lastcolors[8];
struct collTable smash;
```

```
GfxBase = (struct GfxBase *)OpenLibrary("graphics",0);
GfxBase->Debug = 0;
LoadClns(colors);
```

```
writePort = &rP;
InitRastPort(writePort, WIDTH, HEIGHT);
rP.Planes = &pptr[0];
rP.Depth = DEPTH;
for (i = 0; i < DEPTH; i++) pptr[i] = (char *)AllocRaster(WIDTH, HEIGHT);
SetRast(writePort, 0);
InitView(&v);
v.GelsInfo = gi = &gelsinfo;
gi->nextLine = &nextlines[0];    gi->lastColor = &lastcolors[0];
gi->leftmost = gi->topmost = 0;
gi->rightmost = WIDTH - 7;    gi->bottommost = HEIGHT - 1;
gi->collHandler = &smash;    gi->sprRsrvd = -1;
```

```
v.ViewPort = &vp;
viewArgs.DHeight = viewArgs.RHeight = HEIGHT;
viewArgs.DWidth = viewArgs.RWidth = WIDTH;
viewArgs.DxOffset = viewArgs.DyOffset = 0;
viewArgs.RxOffset = viewArgs.RyOffset = 0;
viewArgs.Modes = 0;
InitVPort(&vp, &viewArgs);
vp.Planes = &pptr[0];
vp.Depth = DEPTH;
vp.Mask = 0x3F >> (6 - DEPTH);
vp.Next = NULL;
```

```
NewView(&v);
InitGels(&dumSprite[0], &dumSprite[1]);
SetCollision(BORDERHIT, &borderPatrol);
```

```
sprite.x = 100;    sprite.y = 50;
sprite.vSFlags = SAVEBACK | OVERLAY;
```


CONFIDENTIAL

```

sprite.height = SHEIGHT;      sprite.width = SWIDTH;
sprite.depth = SDEPTH;
sprite.meMask = sprite.hitMask = 1;

/* make a work copy of the image */
sprite.imageData = (short *)getmem(sizeof(image0));
CopyOfLongs(sprite.imageData, image0, sizeof(image0) / sizeof(long));

/* initialize boundary collision mask */
sprite.borderLine = (short *)getmem(4);
sprite.collMask = (short *)getmem(2 * SHEIGHT * SWIDTH);
InitMasks(&sprite);
sprite.sprColors = 0;

/* Selects 1 plane of image as data for Plane 0 of RastPort.*/
sprite.planePick = 0x01;
/* Selects all bits off for other Planes of RastPort.*/
sprite.planeOnOff = 0x00;
vx = 3;          vy = 4;

sprite.vSBob = &bob;
bob.bobVSprite = &sprite;
bob.imageShadow = sprite.collMask;
bob.saveBuffer = (short *)getmem(2 * SHEIGHT * SWIDTH * DEPTH);
bob.savePlanes = &rp[0];
bob.before = bob.after = 0;
bob.bobFlags = 0;
bob.bobComp = 0;
bob.dBuffer = 0;
AddBob(&bob); /* Append bob to display list.*/

MngCop(&v);  WaitBlit();  LoadView(&v);

FOREVER {
    sprite.x += vx; sprite.y += vy;
    SortGList();  DoCollision();  WaitTOF();  DrawGList(writePort); }
}

static short dummy; /* force alignment to even byte at end of file's data */

```

DDT Help Screens

B <bkpt #>,<address><CR>

BREAKPOINTS

Breakpoints are set by typing 'B', the number (1-4), a delimiter, and the address in RAM where you would like the breakpoint set. ROM breakpoints are legal in INTERPRET mode. When a breakpoint is encountered, execution is halted and the screen will update to the address of the breakpoint. To clear an active breakpoint, type 'B <#>,<CR>'. You may update a breakpoint to a new address without clearing the old value first. Breakpointed instructions are shown in inverse video.

B 1,4000 - Set a brkpoint at \$4000.

B 2,> - Set a brkpoint at display address.

C

CONTINUE

The Continue function is a combination of the Breakpoint and Go commands. Issuing a "C" causes a single step past the current instruction, then a breakpoint to be placed at the instruction just executed, then a Go command. When (if) control returns to the address where the breakpoint was left, execution is halted. This command is most useful in loops. The breakpoint used is #0, and may be cleared manually. It is cleared automatically by a user-issued Go command.

D <byte><byte><CR> -or- D 'text<CR>

DEPOSIT

This function deposits bytes into Target memory at the currently displayed address. The "byte" is entered in the form of ASCII nybbles. To deposit \$15, type "D 15". You may enter up to 8 bytes at once. You may also enter bytes in ASCII form. Type "D'Hello<CR>". If you are in the Hex/ASCII window, the displayed position is advanced by the number of bytes that you deposit. If you are in the Instruction window, the display is not advanced.

D 48656C6C6F<CR> -or- D 'Hello<CR>

E <address><CR> -or- E 'label<CR> EXAMINE

Examine allows you to move the display window to any address in memory. The contents of the location are displayed in the current WINDOW mode. You may toggle the current window format with the WINDOW key. To move up and down through memory, use the cursor control keys. Left/Right keys shift the window by a word. Up/Down keys shift the window by a line. PgUp PgDn keys shift the window by a screen.

```
E 4000<CR> - Examine memory at $4000
E 'main<CR> - Examine memory at 'main'
```

G <addr><CR> GO

GO passes control to the Target machine at the specified address. There are two shorthand notations that may be used with GO. The "*" key is an alias for the current PC and a <CR>. The ">" key (unshifted) is an alias for the current window display address and a <CR>.

```
G 4000<CR> - Go at $4000
G *         - Go at current PC
G >        - Go at current display address
```

| | |
|---|-----------|
| I | INTERPRET |
|---|-----------|

INTERPRET has two modes. In the 'High Speed' mode, you have control over the speed of execution, but there is no display updating. In the 'Low Speed' mode, you may select which of the dynamic 'screen areas you wish to have updated. (See TRACE OPTIONS). In either mode, execution continues until :

- ```
* You press a key
* A Trap address is encountered (Low Speed)
* A marked Variable is changed (Low Speed)
```

When execution halts, the screen is updated.

CONFIDENTIAL

M <range, base address><CR>

MONITOR

The MONITOR function allows you to name and monitor specific memory locations in the Target machine. The range parameter defines the range of memory locations from the base address that are considered part of the named location. To define a word variable at 5000, you would enter "M 2,5000<CR>". The cursor is then placed into the Label Window, and you may enter the name of the variable (up to 8 characters). To toggle between addresses and values, type 'M <CR>'.

Range Parameters:

O-Routine      1-Byte      2-Word

P

PROCEDURE CALL

The PROC command deposits a breakpoint at the next instruction in memory after the current one, and issues a Go command. This is useful when you wish to execute subroutines as single steps. Note that you should not use 'P' to execute an RTS type instruction. Be careful when you use it with a Bxx instruction. PROCs don't work in ROM.

ESC

ESCAPE

ESCaping the debugger exits to MS-DOS. The Target is left in its current state. You may leave a program running after executing a GO command, re-enter DDT and stop the program. Symbols defined in the mini-symbol table are lost.

R <register,value>

REGISTER

The "R"egister command allows you to change the contents of the D Registers, the A Registers, the User and Supervisor Stacks, and the PC. Changing the PC allows shorthand notation. '\*' is an alias for 'R PC'.

```
R DO,1234 <CR> - Register DO = 1234
R SS,20000 <CR> - Supervisor Stack = 20000
R SR,2300 <CR> - Status Word = 2300
* > <CR> - PC = current display address
```



## S

## SYNC

The SYNC command causes the Target to resync with the host. The Debugger screen is updated after refreshing all of the dynamic information. This is useful if the Target is out of control, and the NWIRE code has not been smashed. A Interrupt will be issued if the Target does not respond to SYNC request. Then, another SYNC request will be attempted. If the Target is unreachable, the Target ID will be replaced with a flashing 'DEAD!' message.

## T

## TRACE OPTIONS

Setting Trace options allows you to select High or Low speed execution during Interpreted execution. You may select High Speed, which does no display updates or breakpoint checking. Execution speed is then controlled with the left & right cursor keys. Alternatively, you may select any combination of Instruction Window, Stack, Registers and Symbol updating. High Speed and Display Updates are mutually exclusive. Don't use 'Call User Proc' unless you know what you're doing with it.

## V &lt; - &gt; &lt; &gt;

## VARIABLE

You mark a variable by pressing the "V" variable key, which places the cursor in the Variable window. To maneuver to the variable that you want to mark, use the return key. To mark a variable, use the "-" or the "V" keys. To unmark a variable, use the space bar. When a marked variable has been changed, a "\*" replaces the "-". Marked variables changed during a Go, Continue, or JSR command are flagged with the "\*" character, but execution is not halted upon the event. Variables changed in the Interpret mode cause execution to stop.

## W

## WINDOW

Pressing the "W"indow key causes the Display Screen to toggle between the HEX/ASCII window and the Instruction Disassembly window.

Table 1-2. Instruction Set

| Mnemonic | Description                          |
|----------|--------------------------------------|
| ABCD     | Add Decimal with Extend              |
| ADD      | Add                                  |
| AND      | Logical And                          |
| ASL      | Arithmetic Shift Left                |
| ASR      | Arithmetic Shift Right               |
| Bcc      | Branch Conditionally                 |
| BCHG     | Bit Test and Change                  |
| BCLR     | Bit Test and Clear                   |
| BRA      | Branch Always                        |
| BSET     | Bit Test and Set                     |
| BSR      | Branch to Subroutine                 |
| BTST     | Bit Test                             |
| CHK      | Check Register Against Bounds        |
| CLR      | Clear Operand                        |
| CMP      | Compare                              |
| DBcc     | Test Condition, Decrement and Branch |
| DIVS     | Signed Divide                        |
| DIVU     | Unsigned Divide                      |
| EOR      | Exclusive Or                         |
| EXG      | Exchange Registers                   |
| EXT      | Sign Extend                          |
| JMP      | Jump                                 |
| JSR      | Jump to Subroutine                   |
| LEA      | Load Effective Address               |
| LINK     | Link Stack                           |
| LSL      | Logical Shift Left                   |
| LSR      | Logical Shift Right                  |
| MOVE     | Move                                 |
| MOVEM    | Move Multiple Registers              |
| MOVEP    | Move Peripheral Data                 |
| MULS     | Signed Multiply                      |
| MULU     | Unsigned Multiply                    |
| NBCD     | Negate Decimal with Extend           |
| NEG      | Negate                               |
| NOP      | No Operation                         |
| NO       | Ones Complement                      |
| OR       | Logical Or                           |
| PEA      | Push Effective Address               |
| RESET    | Reset External Devices               |
| ROL      | Rotate Left without Extend           |
| ROR      | Rotate Right without Extend          |
| ROXL     | Rotate Left with Extend              |
| ROXR     | Rotate Right with Extend             |
| RTE      | Return from Exception                |
| RTR      | Return and Restore                   |
| RTS      | Return from Subroutine               |
| SBCD     | Subtract Decimal with Extend         |
| Scc      | Set Conditional                      |
| STOP     | Stop                                 |
| SUB      | Subtract                             |
| SWAP     | Swap Data Register Halves            |
| TAS      | Test and Set Operand                 |
| TRAP     | Trap                                 |
| TRAPV    | Trap on Overflow                     |
| TST      | Test                                 |
| UNLK     | Unlink                               |

Hex and Decimal Conversion

|   | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | A   | B   | C   | D   | E   | F   |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  | 12  | 13  | 14  | 15  |
| 1 | 16  | 17  | 18  | 19  | 20  | 21  | 22  | 23  | 24  | 25  | 26  | 27  | 28  | 29  | 30  | 31  |
| 2 | 32  | 33  | 34  | 35  | 36  | 37  | 38  | 39  | 40  | 41  | 42  | 43  | 44  | 45  | 46  | 47  |
| 3 | 48  | 49  | 50  | 51  | 52  | 53  | 54  | 55  | 56  | 57  | 58  | 59  | 60  | 61  | 62  | 63  |
| 4 | 64  | 65  | 66  | 67  | 68  | 69  | 70  | 71  | 72  | 73  | 74  | 75  | 76  | 77  | 78  | 79  |
| 5 | 80  | 81  | 82  | 83  | 84  | 85  | 86  | 87  | 88  | 89  | 90  | 91  | 92  | 93  | 94  | 95  |
| 6 | 96  | 97  | 98  | 99  | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
| 8 | 128 | 129 | 130 | 131 | 132 | 133 | 134 | 135 | 136 | 137 | 138 | 139 | 140 | 141 | 142 | 143 |
| 9 | 144 | 145 | 146 | 147 | 148 | 149 | 150 | 151 | 152 | 153 | 154 | 155 | 156 | 157 | 158 | 159 |
| A | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 | 175 |
| B | 176 | 177 | 178 | 179 | 180 | 181 | 182 | 183 | 184 | 185 | 186 | 187 | 188 | 189 | 190 | 191 |
| C | 192 | 193 | 194 | 195 | 196 | 197 | 198 | 199 | 200 | 201 | 202 | 203 | 204 | 205 | 206 | 207 |
| D | 208 | 209 | 210 | 211 | 212 | 213 | 214 | 215 | 216 | 217 | 218 | 219 | 220 | 221 | 222 | 223 |
| E | 224 | 225 | 226 | 227 | 228 | 229 | 230 | 231 | 232 | 233 | 234 | 235 | 236 | 237 | 238 | 239 |
| F | 240 | 241 | 242 | 243 | 244 | 245 | 246 | 247 | 248 | 249 | 250 | 251 | 252 | 253 | 254 | 255 |

Table 1-1. Data Addressing Modes

68000

| Mode                                                                                                                                                                                             | Generation                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| Register Direct Addressing<br>Data Register Direct<br>Address Register Direct                                                                                                                    | EA = Dn<br>EA = An                                                                                        |
| Absolute Data Addressing<br>Absolute Short<br>Absolute Long                                                                                                                                      | EA = (Next Word)<br>EA = (Next Two Words)                                                                 |
| Program Counter Relative Addressing<br>Relative with Offset<br>Relative with Index and Offset                                                                                                    | EA = (PC) + d16<br>EA = (PC) + (Xn) + dg                                                                  |
| Register Indirect Addressing<br>Register Indirect<br>Postincrement Register Indirect<br>Predecrement Register Indirect<br>Register Indirect with Offset<br>Indexed Register Indirect with Offset | EA = (An)<br>EA = (An), An ← An + N<br>An ← An - N, EA = (An)<br>EA = (An) + d16<br>EA = (An) + (Xn) + dg |
| Immediate Data Addressing<br>Immediate<br>Quick Immediate                                                                                                                                        | DATA = Next Word(s)<br>Inherent Data                                                                      |
| Implied Addressing<br>Implied Register                                                                                                                                                           | EA = PC, USP, SP, PC                                                                                      |

## NOTES:

EA = Effective Address

An = Address Register

Dn = Data Register

Xn = Address or Data Register used as Index Register

SR = Status Register

PC = Program Counter

dg = 8-bit Offset (displacement)

d16 = 16-bit Offset (displacement)

N = 1 for Byte, 2 for Words, and 4 for Long Words

If An is the stack pointer and the operand size is

byte, N = 2 to keep the stack pointer on a word

boundary.

( ) = Contents of

← = Replaces

CC — Carry Clear  
CS — Carry Set  
EQ — Equal  
F — Never True  
GE — Greater or Equal  
GT — Greater Than  
HI — High  
LE — Less or Equal  
LS — Low or Same  
LT — Less Than  
MI — Minus  
NE — Not Equal  
PL — Plus  
T — Always True  
VC — No Overflow  
VS — Overflow

Table 1-3. Variations of Instruction Types

| Instruction Type | Variation                                                                          | Description                                                                                                                                         |
|------------------|------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| ADD              | ADD<br>ADDA<br>ADDQ<br>ADDI<br>ADDX                                                | Add<br>Add Address<br>Add Quick<br>Add Immediate<br>Add with Extend                                                                                 |
| AND              | AND<br>ANDI<br>ANDI to CCR<br>ANDI to SR                                           | Logical AND<br>AND Immediate<br>AND Immediate to Condition Code<br>AND Immediate to Status Register                                                 |
| CMP              | CMP<br>CMPA<br>CMPM<br>CMPI                                                        | Compare<br>Compare Address<br>Compare Memory<br>Compare Immediate                                                                                   |
| EOR              | EOR<br>EORI<br>EORI to CCR<br>EORI to SR                                           | Exclusive OR<br>Exclusive OR Immediate<br>Exclusive Immediate to Condition Codes<br>Exclusive OR Immediate to Status Register                       |
| MOVE             | MOVE<br>MOVEA<br>MOVEQ<br>MOVE to CCR<br>MOVE to SR<br>MOVE from SR<br>MOVE to USP | Move<br>Move Address<br>Move Quick<br>Move to Condition Codes<br>Move to Status Register<br>Move from Status Register<br>Move to User Stack Pointer |
| NEG              | NEG<br>NEGX                                                                        | Negate<br>Negate with Extend                                                                                                                        |
| OR               | OR<br>ORI<br>ORI to CCR<br>ORI to SR                                               | Logical OR<br>OR Immediate<br>OR Immediate to Condition Codes<br>OR Immediate to Status Register                                                    |
| SUB              | SUB<br>SUBA<br>SUBI<br>SUBQ<br>SUBX                                                | Subtract<br>Subtract Address<br>Subtract Immediate<br>Subtract Quick<br>Subtract with Extend                                                        |

ASCII

|     | MSD  | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   |
|-----|------|-----|-----|-----|-----|-----|-----|-----|-----|
| LSD | 000  | 001 | 010 | 011 | 100 | 101 | 110 | 111 |     |
| 0   | 0000 | NUL | DC1 | 1   | 0   | @   | P   |     |     |
| 1   | 0001 | SOH | DC1 | 1   | 1   | A   | O   | a   | q   |
| 2   | 0010 | STX | DC2 | 2   | 2   | B   | R   | b   | r   |
| 3   | 0011 | ETX | DC3 | 3   | 3   | C   | S   | c   | s   |
| 4   | 0100 | EOT | DC4 | 4   | 4   | D   | T   | d   | t   |
| 5   | 0101 | END | NAK | 5   | 5   | E   | U   | e   | u   |
| 6   | 0110 | ACA | SYN | 6   | 6   | F   | V   | v   |     |
| 7   | 0111 | BEL | ETB | 7   | 7   | G   | W   | w   |     |
| 8   | 1000 | BS  | CAN | 8   | 8   | H   | X   | x   |     |
| 9   | 1001 | HT  | EW  | 9   | 9   | I   | Y   | y   |     |
| A   | 1010 | LF  | SUB |     |     | J   | Z   | z   |     |
| B   | 1011 | VT  | ESC |     |     | K   | [   |     |     |
| C   | 1100 | FF  | FS  |     |     | L   | \   |     |     |
| D   | 1101 | CR  | GS  |     |     | M   | ]   |     |     |
| E   | 1110 | SO  | RS  |     |     | N   | ^   |     |     |
| F   | 1111 | SI  | US  |     |     | O   | _   |     | DEL |

Table 5-2. Exception Vector Assignment

| Vector Number(s)   | Dec  | Address Hex | Space | Assignment                            |
|--------------------|------|-------------|-------|---------------------------------------|
| 0                  | 0    | 000         | SP    | Reset: Initial SSP <sup>2</sup>       |
|                    | 4    | 004         | SP    | Reset: Initial PC <sup>2</sup>        |
| 2                  | 8    | 008         | SD    | Bus Error                             |
| 3                  | 12   | 00C         | SD    | Address Error                         |
| 4                  | 16   | 010         | SD    | Illegal Instruction                   |
| 5                  | 20   | 014         | SD    | Zero Divide                           |
| 6                  | 24   | 018         | SD    | CHK Instruction                       |
| 7                  | 28   | 01C         | SD    | TRAPV Instruction                     |
| 8                  | 32   | 020         | SD    | Privilege Violation                   |
| 9                  | 36   | 024         | SD    | Trace                                 |
| 10                 | 40   | 028         | SD    | Line 1010 Emulator                    |
| 11                 | 44   | 02C         | SD    | Line 1111 Emulator                    |
| 12 <sup>1</sup>    | 48   | 030         | SD    | (Unassigned, Reserved)                |
| 13 <sup>1</sup>    | 52   | 034         | SD    | (Unassigned, Reserved)                |
| 14 <sup>1</sup>    | 56   | 038         | SD    | (Unassigned, Reserved)                |
| 15                 | 60   | 03C         | SD    | Uninitialized Interrupt Vector        |
| 16-23 <sup>1</sup> | 64   | 040         | SD    | (Unassigned, Reserved)                |
|                    | 95   | 05F         |       | —                                     |
| 24                 | 96   | 060         | SD    | Spurious Interrupt <sup>3</sup>       |
| 25                 | 100  | 064         | SD    | Level 1 Interrupt Autovector          |
| 26                 | 104  | 068         | SD    | Level 2 Interrupt Autovector          |
| 27                 | 108  | 06C         | SD    | Level 3 Interrupt Autovector          |
| 28                 | 112  | 070         | SD    | Level 4 Interrupt Autovector          |
| 29                 | 116  | 074         | SD    | Level 5 Interrupt Autovector          |
| 30                 | 120  | 078         | SD    | Level 6 Interrupt Autovector          |
| 31                 | 124  | 07C         | SD    | Level 7 Interrupt Autovector          |
| 32-47              | 128  | 080         | SD    | TRAP Instruction Vectors <sup>4</sup> |
|                    | 191  | 0BF         |       | —                                     |
| 48-63 <sup>1</sup> | 192  | 0C0         | SD    | (Unassigned, Reserved)                |
|                    | 255  | 0FF         |       | —                                     |
| 64-255             | 256  | 100         | SD    | User Interrupt Vectors                |
|                    | 1023 | 3FF         |       | —                                     |

- NOTES:
1. Vector numbers 12, 13, 14, 16 through 23, and 48 through 63 are reserved for future enhancements by Motorola. No user peripheral devices should be assigned these numbers.
  2. Reset vector (0) requires four words, unlike the other vectors which only require two words, and is located in the supervisor program space.
  3. The spurious interrupt vector is taken when there is a bus error indication during interrupt processing. Refer to Paragraph 5.5.2.
  4. TRAP #n uses vector number 32 + n.

Table A-1. Condition Code Computations

| Operations                                                          | X | N | Z | V | C | Special Definition                                                                                                                                     |
|---------------------------------------------------------------------|---|---|---|---|---|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| ABCD                                                                | * | U | ? | U | ? | C = Decimal Carry<br>$Z = Z \cdot Rm \cdot \dots \cdot R0$                                                                                             |
| ADD, ADDI, ADDQ                                                     | * | * | * | ? | ? | $V = Sm \cdot Dm \cdot Rm + Sm \cdot \overline{Dm} \cdot Rm$<br>$C = Sm \cdot Dm + Rm \cdot Dm + Sm \cdot Rm$                                          |
| ADDX                                                                | * | * | ? | ? | ? | $V = Sm \cdot Dm \cdot Rm + Sm \cdot \overline{Dm} \cdot Rm$<br>$C = Sm \cdot Dm + Rm \cdot Dm + Sm \cdot Rm$<br>$Z = Z \cdot Rm \cdot \dots \cdot R0$ |
| AND, ANDI, EOR, EORI, MOVEQ, MOVE, OR, ORI, CLR, EXT, NOT, TAS, TST | — | * | * | 0 | 0 |                                                                                                                                                        |
| CHK                                                                 | — | * | U | U | U |                                                                                                                                                        |
| SUB, SUBI, SUBQ                                                     | * | * | * | ? | ? | $V = Sm \cdot Dm \cdot Rm + Sm \cdot \overline{Dm} \cdot Rm$<br>$C = Sm \cdot Dm + Rm \cdot Dm + Sm \cdot Rm$                                          |
| SUBX                                                                | * | * | ? | ? | ? | $V = Sm \cdot Dm \cdot Rm + Sm \cdot \overline{Dm} \cdot Rm$<br>$C = Sm \cdot Dm + Rm \cdot Dm + Sm \cdot Rm$<br>$Z = Z \cdot Rm \cdot \dots \cdot R0$ |
| CMP, CMPI, CMPM                                                     | — | * | * | ? | ? | $V = Sm \cdot Dm \cdot Rm + Sm \cdot \overline{Dm} \cdot Rm$<br>$C = Sm \cdot Dm + Rm \cdot Dm + Sm \cdot Rm$                                          |
| DIVS, DIVU                                                          | — | * | * | ? | 0 | V = Division Overflow                                                                                                                                  |
| MULS, MULU                                                          | — | * | * | 0 | 0 |                                                                                                                                                        |
| SBCD, NBCD                                                          | * | U | ? | U | ? | C = Decimal Borrow<br>$Z = Z \cdot Rm \cdot \dots \cdot R0$                                                                                            |
| NEG                                                                 | * | * | * | ? | ? | $V = Dm \cdot Rm, C = Dm + Rm$                                                                                                                         |
| NEGX                                                                | * | * | ? | ? | ? | $V = Dm \cdot Rm, C = Dm + Rm$<br>$Z = Z \cdot Rm \cdot \dots \cdot R0$                                                                                |
| BTST, BCHG, BSET, BCLR                                              | — | — | ? | — | — | $Z = \overline{Dn}$                                                                                                                                    |
| ASL                                                                 | * | * | * | ? | ? | $V = Dm \cdot (Dm - 1 + \dots + Dm - r)$<br>$+ Dm \cdot (Dm - 1 + \dots + Dm - r)$<br>$C = Dm - r + 1$                                                 |
| ASL (r=0)                                                           | — | * | * | 0 | 0 |                                                                                                                                                        |
| LSL, ROXL                                                           | * | * | * | 0 | ? | $C = Dm - r + 1$                                                                                                                                       |
| LSR (r=0)                                                           | — | * | * | 0 | 0 |                                                                                                                                                        |
| ROXL (r=0)                                                          | — | * | * | 0 | ? | $C = X$                                                                                                                                                |
| ROL                                                                 | — | * | * | 0 | ? | $C = Dm - r + 1$                                                                                                                                       |
| ROL (r=0)                                                           | — | * | * | 0 | 0 |                                                                                                                                                        |
| ASR, LSR, ROXR                                                      | * | * | * | 0 | ? | $C = Dr - 1$                                                                                                                                           |
| ASR, LSR (r=0)                                                      | — | * | * | 0 | 0 |                                                                                                                                                        |
| ROXR (r=0)                                                          | — | * | * | 0 | ? | $C = X$                                                                                                                                                |
| ROR                                                                 | — | * | * | 0 | ? | $C = Dr - 1$                                                                                                                                           |
| ROR (r=0)                                                           | — | * | * | 0 | 0 |                                                                                                                                                        |

— Not affected  
U Undefined  
? Other — see Special Definition

\* General Case:  
 $X = C$   
 $N = Rm$   
 $Z = Rm \cdot \dots \cdot R0$

Sm Source Operand — most significant bit  
Dm Destination operand — most significant bit  
Rm Result operand — most significant bit  
n bit number  
r shift count

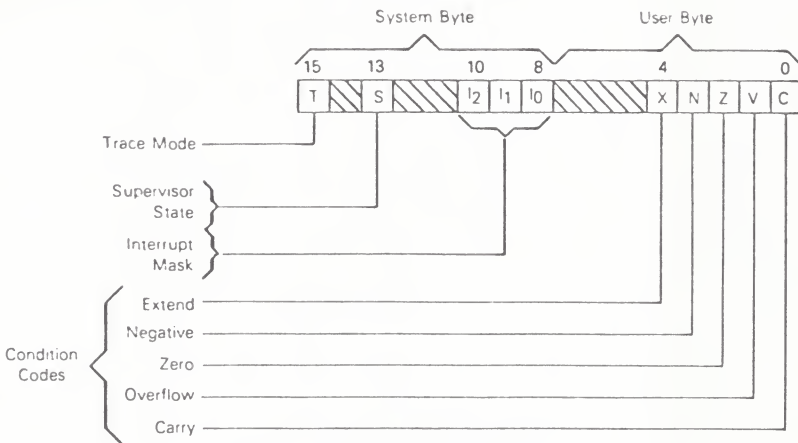


Figure 1-3. Status Register



where:

- N (negative) Set if the most significant bit of the result is set. Cleared otherwise.  
Z (zero) Set if the result equals zero. Cleared otherwise.  
V (overflow) Set if there was an arithmetic overflow. This implies that the result is not representable in the operand size. Cleared otherwise.  
C (carry) Set if a carry is generated out of the most significant bit of the operands for an addition. Also set if a borrow is generated in a subtraction. Cleared otherwise.  
X (extend) Transparent to data movement. When affected, it is set the same as the C bit.

The notational convention that appears in the representation of the condition code register is:

- \* set according to the result of the operation  
— not affected by the operation  
0 cleared  
1 set  
U undefined after the operation

**brought to you by  
andy finkel**